Specifying digital circuits

- Schematics (what we’ve done so far)
  - Structural description
  - Describe circuit as interconnected elements
    - Build complex circuits using hierarchy
    - Large circuits are unreadable
- Hardware description languages (HDLs)
  - Structural and behavioral descriptions
    - Not programming languages
    - They are parallel languages tailored to digital design
  - Synthesize code to produce a circuit from descriptions
    - Easier to modify designs
    - Details of realization take care of by compiler

Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics
Specifying circuits in Verilog

- There are three major styles
  - Instances 'n wires
  - Continuous assignments
  - “always” blocks

```
wire E;
and g1(E,A,B);
not g2(Y,C);
or  g3(X,E,Y);
```

```
wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
```

---

Modules are circuit components

- Module has ports
  - External connections
  - A, B, C, X, Y in this example
- Port types
  - input (A, B, C)
  - output (X, Y)
  - inout (tri-state) – more later
- Use assign statements for
  Boolean expressions
  - and ⇔ &
  - or ⇔ |
  - not ⇔ ~

```
module smpl2 (X,Y,A,B,C);
  input A,B,C;
  output X,Y;
  assign X = (A&B)|~C;
  assign Y = ~C;
endmodule
```
Example: A comparator

```
module Compare1 (Equal, Alarger, Blarger, A, B);
input     A, B;
output    Equal, Alarger, Blarger;

assign Equal   = (A & B) | (~A & ~B);
assign Alarger = (A & ~B);
assign Blarger = (~A & B);
endmodule
```

assign statement ordering doesn't matter because they execute in parallel

Preferred (Verilog 2001) Port Declaration Style

```
module Compare1 (output Equal, Alarger, Blarger, A, B);
input     A, B;
output    Equal, Alarger, Blarger;

assign Equal   = (A & B) | (~A & ~B);
assign Alarger = (A & ~B);
assign Blarger = (~A & B);
endmodule
```
>1 **assign** to the Same Variable is Illegal

```verilog
assign A = X | (Y & ~Z);
assign B = W | A;
assign A = Y & Z;
```

"Reusing" a variable on the LHS in multiple assign statements is not allowed

---

**Verilog for Sequential Logic**

- Registers implemented using "always" block
  - `@`(posedge clk) means that the statement is only executed on the positive edge of the clock
  - i.e. input `d` is sampled, and assigned to `q` only when the clock ticks
  - Note "reg" declaration – needed for always block assignment
  - Does **NOT** mean register!!

```verilog
module dff (clk, d, q);
    input clk, d;
    output q;
    reg q;
    always @(posedge clk)
        q <= d;
endmodule
```
Combining Combinational and Sequential Logic

Example: Enabled d flip-flop

```verilog
module dff
    (input clk,
     input d,
     input cen, // Clock enable
     output reg q);

    wire qNext;
    assign qNext = cen ? d : q;

    always @(posedge clk)
        q <= qNext;
endmodule
```

Example: 8-bit register with synchronous reset

```verilog
module reg8
    (input clk,
     input reset, // Synchronous reset
     input [7:0] d,
     output reg [7:0] q);

    wire [7:0] qNext;
    assign qNext = reset ? 0 : d;

    always @(posedge clk)
        q <= qNext;
endmodule
```
Combining Combinational and Sequential Logic

- Example: 8-bit shift register

```verilog
module shift8
  (input clk,
   input [1:0] control, // shift control
   input [7:0] d, // parallel input
   output reg [7:0] q);

  wire [7:0] qNext;
  assign qNext =
    control == 0 ? 0 : // reset
                 control == 1 ? d :  // load
                 control == 2 ? {q[6:0], q[7]} : // shift left
                    /* control == 3*/ {q[0], q[7:1]}; // shift right

  always @(posedge clk)
    q <= qNext;
endmodule
```

always Block Behavioral Verilog

- C-like statements in the always block
- Describe what the circuit should do
  - Shorthand for register + combinational logic

```verilog
module shift8
  (input clk,
   input [1:0] control, // shift control
   input [7:0] d, // parallel input
   output reg [7:0] q);

  always @(posedge clk)
    case (control)
    0 : q <= 0;   // reset
    1 : q <= d;   // load
    2 : q <= {q[6:0], q[7]}; // shift left
    3 : q <= {q[0], q[7:1]}; // shift right
    default : q <= 0;
    endcase
endmodule
```
reset : Synchronous and Asynchronous

- C-like statements in the always block
- Describe what the circuit should do
  - Shorthand for register + combinational logic

```verilog
// Synchronous reset
module reg8
  (input clk,
   input reset,
   input [7:0] d,
   output reg [7:0] q);

always @ (posedge clk)
  if (reset) q <= 0;
else q <= d;
endmodule
```

```verilog
// Asynchronous reset (high)
module reg8
  (input clk,
   input reset,
   input [7:0] d,
   output reg [7:0] q);

always @ (posedge clk or
            posedge reset)
  if (reset) q <= 0;
else q <= d;
endmodule
```

Describing Hierarchy – Ripple-Carry Adder Example

```verilog
// 1-bit adder
module add1
  (input a,
   input b,
   input cin,
   output sum,
   output cout);

// Add 3 1-bit numbers with a 2-bit result
assign { cout, sum } = a + b + cin;
endmodule
```
Ripple-Carry Adder Example

// 4-bit adder
module add4
    (input [3:0] a,
     input [3:0] b,
     input cin,
     output [3:0] sum,
     output cout);
    wire c1, c2, c3;
    add1 a1_0 (.a (a[0]), .b (b[0]), .cin (cin), .cout (c1), .sum (sum[0]));
    add1 a1_1 (.a (a[1]), .b (b[1]), .cin (c1), .cout (c2), .sum (sum[1]));
    add1 a1_2 (.a (a[2]), .b (b[2]), .cin (c2), .cout (c3), .sum (sum[2]));
    add1 a1_3 (.a (a[3]), .b (b[3]), .cin (c3), .cout (cout), .sum (sum[3]));
endmodule

Ripple-Carry Adder Example

// 16-bit adder
module add16
    (input [15:0] a,
     input [15:0] b,
     input cin,
     output [15:0] sum,
     output cout);
    wire c1, c2, c3;
    add4 a4_0 (.a (a[3:0]), .b (b[3:0]), .cin (cin), .cout (c1), .sum (sum[3:0]));
    add4 a4_1 (.a (a[7:4]), .b (b[7:4]), .cin (c1), .cout (c2), .sum (sum[7:4]));
    add4 a4_2 (.a (a[11:8]), .b (b[11:8]), .cin (c2), .cout (c3), .sum (sum[11:8]));
    add4 a4_3 (.a (a[15:12]), .b (b[15:12]), .cin (c3), .cout (cout), .sum (sum[15:12]));
endmodule
Describing Circuit Hierarchy

- Like a file system hierarchy

**Definition Hierarchy**

- add16
  - 4 instances of add4
- add4
  - 4 instances of add1

**Instantiation Hierarchy**

- add16 (a16)
  - add4 (a4_0)
    - add1 (a1_0)
    - add1 (a1_1)
    - add1 (a1_2)
    - add1 (a1_3)
  - add4 (a4_1)
    - add1 (a1_0)
    - add1 (a1_1)
    - add1 (a1_2)
    - add1 (a1_3)
  - add4 (a4_2)
    - add1 (a1_0)
    - add1 (a1_1)
    - add1 (a1_2)
    - add1 (a1_3)
  - add4 (a4_3)
    - add1 (a1_0)
    - add1 (a1_1)
    - add1 (a1_2)
    - add1 (a1_3)

Behavioral Verilog

- C-like statements inside always block
- if and case
  - Very useful for describing conditional logic
  - Generally compiles into multiplexers
- always @(posedge clk) begin
  statement1
  statement2
  etc.
end
  - Statements are executed sequentially to describe
    the intended behavior of the circuit
  - Last assignment to variable is value put into the register
- Use with caution!
Some Details: Verilog Data Types and Values

- **Bits** - value on a wire
  - 0, 1
  - X - don’t care
  - Z - undriven, tri-state
- **Vectors of bits**
  - Treated as an *unsigned* integer value by default
    - e.g. `A < 0` is *never* true!
  - Concatenating bits/vectors into a vector
    - e.g. sign extend
- Vectors can be declared as "signed"

Numbers in Verilog

- **14** - ordinary decimal number
- **-14** - 2’s complement representation
- **12’h0000_0100_0110** - binary number with 12 bits (_ is ignored)
- **12’h046** - hexadecimal number with 12 bits

Verilog values are *unsigned*

- if `A = 0110` (6) and `B = 1010`(-6)
  - `C = 10000` not `00000`
  - i.e. B is zero-padded, not sign-extended (unless declared signed)
Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bit-select or</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>parenthasis</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
</tr>
<tr>
<td>^</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>.</td>
<td>reduction NOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction XNOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>(</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>[ ]</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
</tbody>
</table>

| >                | greater than    | Relational       |
| >=               | greater than or equal to | Relational       |
| <                | less than       | Relational       |
| <=               | less than or equal to | Relational       |
| ==               | logical equality| Equality         |
| /=               | logical inequality| Equality        |
| ==match          | case equality   | Equality         |
| /=match          | case inequality | Equality         |
| &                | bit-wise AND    | Bit-wise         |
| ^                | bit-wise XOR    | Bit-wise         |
| ^, ^, ^          | bit-wise XNOR   | Bit-wise         |
| |                | bit-wise OR     | Bit-wise         |
| &.&              | logical AND     | Logical          |
| ||                | logical OR      | Logical          |
| ?:                | conditional     | Conditional      |

Verilog Variables

- **wire**
  - variable used to connect components together
  - wires are **assigned** to

- **reg**
  - variable that saves a value as part of a behavioral description
  - may or may not be a register in the circuit
  - regs are assigned to in an always block

- The rule:
  - Declare a variable a reg if it is assigned to in an **always** block
  - wire/reg distinction is confusing and unnecessary
    - but we have to live with it